ELINDA LARA

PYTHON FOR BEGINNERS

A QUICK COURSE FOR MASTERING THE BASICS OF PYTHON

ELINDA LARA

ELINDA LARA

PYTHON FOR BEGINNERS

A QUICK COURSE FOR MASTERING THE BASICS OF PYTHON

ELINDA LARA

Python For Beginners

Table of Contents

Table of Contents

Introduction

Chapter 1: overview

Chapter 2: Environmental Setup

Chapter 3: Basic Syntax

Chapter 4: Variables

Chapter 5: operators

Chapter 6: decision Making

Chapter 7: Loops

Chapter 8: Functions

Chapter 9: Modules

Conclusion

INtroductioN

PythoN is a high-level prograMMiNg laNguage that is iNterpreted, Not coMpiled! Besides this, it is aN oBJect-orieNted prograMMiNg laNguage that supports iNteractioN BetweeN the developer aNd iNterpreter used for iNterpretiNg PythoN code. This prograMMiNg laNguage is kNowN to have BeeN created BetweeN 1985-1990. guido vaN RossuM is credited for this developMeNt.

Just like Perl prograMMiNg laNguage, the source code for PythoN is availaBle uNder gNu geNeral PuBlic LiceNse (gpl). This Book is a BegiNNer's guide to learNiNg PythoN aNd shall help you create the right fouNdatioN for learNiNg advaNced PythoN prograMMiNg.

Moreover, this Book has BeeN desigNed keepiNg the requireMeNts of BegiNNers with No PythoN BackgrouNd iN MiNd. Therefore, you caN start right away aNd learN PythoN froM scratch, with the help of this Book. With that said, it is crucial to MeNtioN here that soMe Basic kNowledge of prograMMiNg aNd prograMMiNg coNcepts shall help you iMMeNsely iN facilitatiNg rapid learNiNg.

We hope you will eNJoy this Book. happy codiNg!

Copyright 2017 - All rights reserved.

This docuMeNt is geared towards providiNg exact aNd reliaBle iNforMatioN iN regards to the topic aNd issue covered. The puBlicatioN is sold with the idea that the puBlisher is Not required to reNder accouNtiNg, officially perMitted, or otherwise, qualified services. If advice is Necessary, legal or professioNal, a practiced iNdividual iN the professioN should Be ordered.

- FroM a declaratioN of PriNciples which was accepted aNd approved equally By a coMMittee of the AMericaN Bar AssociatioN aNd a coMMittee of PuBlishers aNd AssociatioNs.

IN No way is it legal to reproduce, duplicate, or traNsMit aNy part of this docuMeNt iN either electroNic MeaNs or iN priNted forMat. RecordiNg of this puBlicatioN is strictly prohiBited aNd aNy storage of this docuMeNt is Not allowed uNless with writteN perMissioN froM the puBlisher. All rights reserved.

The iNforMatioN provided hereiN is stated to Be truthful aNd coNsisteNt, iN that aNy liaBility, iN terMs of iNatteNtioN or otherwise, By aNy usage or aBuse of aNy

Policies, processes, or directioNs coNtaiNed withiN is the solitary aNd utter respoNsiBility of the recipieNt reader. UNder No circuMstaNces will aNy legal respoNsiBility or BlaMe Be held againSt the puBlisher for aNy reparatioN, daMages, or MoNetary loss due to the iNforMatioN hereiN, either directly or iNdirectly.

Respective authors owN all copyrights Not held By the puBlisher.

The iNforMatioN hereiN is offered for iNforMatioNal purposes solely, aNd is uNiversal as so. The preseNtatioN of the iNforMatioN is without coNtract or aNy type of guaraNtee assuraNce.

The tradeMarks that are used are without aNy coNseNt, aNd the puBlicatioN of the tradeMark is without perMissioN or BackiNg By the tradeMark owNer. All tradeMarks aNd BraNds withiN this Book are for clarifyiNg purposes oNly aNd are the owNed By the owNers theMselves, Not affiliated with this docuMeNt.

Chapter 1: overview

PythoN is aN iNterpreted, high level, oBJect orieNted laNguage that is coMMoNly used for scriptiNg. The driviNg force BehiNd the developMeNt of PythoN was to create a prograMMiNg laNguage that caN Be easily uNderstood. Therefore, readaBility was oNe of the priMary requireMeNts of the PythoN proJect.

It is Because of this reasoN that PythoN Makes exteNsive use of ENglish keywords without placiNg Much eMphasis oN syNtactical coNstructioNs aNd puNctuatioN. Before we Move oN to actual prograMMiNg, let us look at a little theory aBout the PythoN prograMMiNg laNguage.

history

PythoN is kNowN to have BeeN developed at NatioNal Research INstitute for MatheMatics aNd coMputer ScieNce iN the NetherlaNds arouNd late eighties aNd early NiNeties. guido vaN RossuM is credited for its developMeNt, which is coNsidered to have BeeN iNspired froM MaNy prograMMiNg laNguages like c, c++, aBc, Modula-3, SMalltalk aNd Algol-68.

It is also kNowN to have derived eleMeNts froM the UNix Shell aNd soMe other scriptiNg laNguages. Although, the core developMeNt teaM still MaNages the developMeNt of the laNguage aNd it reMaiNs copyrighted, the source code is Now availaBle as part of gNu geNeral PuBlic LiceNse (gpl).

Key Features

The key features of the PythoN prograMMiNg laNguage iNclude –

- PythoN code is directly iNterpreted By the iNterpreter at ruN-tiMe. Therefore, there is No Need for the prograMMer to coMpile the code. This is also the MaNNer iN which this prograMMiNg laNguage is siMilar to php aNd Perl.
- PythoN allows the developer to iNteract with the iNterpreter at aNy tiMe of executioN. This Makes developMeNt Much siMpler aNd provides uNMatched flexiBility froM the developer's perspective.
- PythoN coMpletely supports aNd iMpleMeNts the oBJect-orieNted Methodology By eNcapsulatiNg code iNside oBJects.
-

PythoN is coNsidered oNe of the easiest laNguages to learN for BegiNNers. however, the siMplicity of the laNguage does Not affect the powerful applicatioNs of the saMe. PythoN caN Be used to create siMple text processiNg applicatioNs as well as gaMes aNd Browsers.

Other features of PythoN have BeeN descriBed Below

– *Easy to read, learN aNd MaiNtaiN*

The preseNce of siMple structures, clearly-defiNed syNtax aNd ENglish-laNguage keywords Makes PythoN a easy to learN, read aNd MaiNtaiN laNguage.

AvailaBility of StaNdard LiBrary

The staNdard liBrary availaBle as part of the PythoN package is cross-platforM aNd highly portaBle. So, irrespective of the platforM you decide to work oN, Be it MaciNtosh, Microsoft WiNdows or LiNux/UNix, you should Not face aNy proBleMs iN usiNg the iNBuilt fuNctioNality.

hardware PortaBility

PythoN supports operatioN oN Multiple hardware platforMs. Moreover, the iNterface for all the differeNt hardware platforMs is the saMe. Therefore, MiNiMal effort is required oN the prograMMer's eNd to shift froM oNe hardware platforM to aNother.

Allows INteractioN BetweeN developer aNd INterpreter

PythoN code caN Be tested aNd deBugged usiNg aN iNteractive Mode of operatioN that allows the prograMMer to coMMuNicate with the iNterpreter.

ScalaBle

UNlike shell scriptiNg that oNly supports sMall prograMs, PythoN eNaBles prograMMers to write Big prograMs with ease.

gui PrograMMiNg

gui applicatioNs usually require the use of systeM calls aNd liBrary fuNctioNs. The aBility of PythoN to support aNd work well with hardware of differeNt coNfiguratioNs allows prograMMers to write portaBle gui applicatioNs.

dataBases

PythoN allows prograMMers to use a aNy of the Multiple dataBases supported By it. IN fact, PythoN supports alMost all coMMercial dataBases.

ExteNdaBle

SMall Modules caN Be added to the PythoN iNterpreter. IN this MaNNer, PythoN allows custoMizatioN of its tools for higher efficieNcy.

Other Features

IN additioN to the aBove-MeNtioNed, PythoN also has soMe other good features that are worth MeNtioNiNg. These iNclude –

- IN additioN to oBJect-orieNted prograMMiNg, PythoN also supports structured aNd procedural prograMMiNg.
- PythoN caN Be used as a staNdard scriptiNg laNguage as well as a prograMMiNg laNguage for large applicatioNs By coMpiliNg prograMs to Byte-code.
- garBage is autoMatically collected By the PythoN ecosysteM.
- PythoN caN easily Be iNtegrated with other prograMMiNg laNguages like C, C++, corBa, Java aNd coM.
- Type checkiNg is doNe dyNaMically aNd PythoN supports MaNy high level data types.

Chapter 2: ENviroNMeNtal Setup

As we MeNtioNed previously, PythoN is availaBle oN a wide raNge of hardware aNd software platforMs. Therefore, you caN work oN PythoN prograMMiNg irrespective of the hardware or software platforM that you are workiNg oN. however, you will have to set up a workiNg eNviroNMeNt for the saMe. This chapter illustrates the Method for settiNg up a local eNviroNMeNt for PythoN prograMMiNg.

Local Setup

If you are Not sure whether your systeM has pythoN already iNstalled or Not, you caN siMply opeN the terMiNal or coMMaNd proMpt aNd type pythoN. If pythoN is already preseNt oN the systeM, this coMMaNd will also retturN the versioN of pythoN that is there. PythoN is supported oN the followiNg operatiNg systeMs –

- WiN 9x/Nt/2000
- UNix (LiNux, Solaris, AIX, freeBsd, suNos, hp/UX aNd IRIX, iN additioN to soMe others)
- Os/2
- MaciNtosh (ppc, INtel aNd 68K)
- PalMos
- dos (Multiple versioNs)
- WiNdows ce
- Nokia MoBile phoNes
- Beos
- AcorN/risc os
- VMS/opeNvMs
- AMiga

- Vxworks
- QNX

- PsioN

IN additioN to the aBove-MeNtioNed, portiNg of PythoN to .Net virtual MachiNes aNd Java has also BeeN doNe.

getting PythoN

Although, pythoN is copyrighted, its source code is availaBle uNder the geNeral gNu liceNse. Therefore, the latest BiNaries aNd source code caN Be directly dowNloaded froM the PythoN weBsite. Besides this, you will also fiNd the docuMeNtatioN related to PythoN aNd prograMMiNg oN the saMe source. The docuMeNtatioN is availaBle iN htMl, postscript as well as pdf forMats.

INstalliNg PythoN

INstalliNg PythoN is as siMple as Just dowNloadiNg the BiNaries for the platforM aNd iNstalliNg PythoN for the saMe. however, if you are usiNg a platforM for which the BiNaries of PythoN are Not availaBle. You caN dowNload the source code aNd coMpile the saMe usiNg a c coMpiler oN the coNcerNed platforM MaNually.

This MaNual iNstallatioN also gives you Much More flexiBility iN terMs of the features that you caN iNcorporate iN your custoM iNstallatioN. This sectioN covers PythoN iNstallatioN for MaiN operatiNg systeMs like uNix, LiNux, WiNdows aNd MaciNtosh.

UNix/LiNux

IN order to iNstall PythoN oN UNix/LiNux, you Need to follow the steps giveN Below –

- go to the official page of PythoN usiNg the weB Browser.
- dowNload the BiNaries availaBle for UNix/LiNux platforM.
- Extract the files to a folder.
- If you wish to custoMize the iNstallatioN By addiNg or reMoviNg soMe optioNs, access the Setup file aNd do the saMe.
- RuN the ./coNfigure script.

give the coMMaNds Make aNd Make iNstall.

These commands shall install Python to */usr/local/bin*. The libraries of python can be found at */usr/local/lib/pythonxx* where XX is the version of Python that you have downloaded.

Windows

In order to install Python on a Windows system, you need to follow the steps given below –

- go to the official page of Python using the web browser.
- download the Windows installer python-XYZ.msi. In order to run this file, please ensure that your Windows system has Microsoft Installer 2.0 installed. Also, note that XYZ is the version of Python you are installing.
- download and save the installer on the local machine and run it.
- Upon running, the file will take you to the Python install wizard and from here onwards, you just need to follow the instructions until the wizard says that the installation is done.

Macintosh

Most of the recent Mac machines have Python pre-installed in them. With that said, they may not have the most recent version and it is always good to check and get the latest version of the language before getting started with Python programming. You can get the current version of Python for Mac along with instructions to install it by accessing the downloads section of the Python website.

Setting the path

The executable files and programs related to the installation may be placed at different locations. This makes it difficult for the system to track which path needs to be accessed when. In order to solve this issue, the os keeps a search path with it, which it uses for searching executable files, and is saved in an environment variable. By communicating this variable, the information is made available to the command shell and other programs that may require it.

This variable is known by the name path in Unix and Path in Windows. Please mind the difference in the case of the letters. This different is present due to the fact that Unix/Linux is case-sensitive and exactly the name path

Must Be used. however, WiNdows is case-iNseNsitive aNd as loNg as the striNg is path, the operatiNg systeM will Be fiNe with it. Lastly, iN case of MaciNtosh, the path details are autoMatically haNdled. The PythoN directory Must Be added to your path for iNvokiNg the PythoN iNterpreter.

UNix/LiNux

IN order to add the PythoN directory for a specific sessioN iN the UNix/LiNux operatiNg systeM, you Need to follow oNe of the three Methods giveN Below –

SettiNg Path UsiNg csh Shell

IN order to set Path usiNg this Method, you Just Need to type the coMMaNd giveN Below aNd type eNter –

Path "$path:/usr/local/BiN/pythoN"

SettiNg Path UsiNg Bash Shell

IN order to set Path usiNg this Method, you Just Need to type the coMMaNd giveN Below aNd type eNter –

Export Ath="$path:/usr/local/BiN/pythoN"

SettiNg Path UsiNg Sh or ksh Shell

IN order to set Path usiNg this Method, you Just Need to type the coMMaNd giveN Below aNd type eNter –

Path="$path:/usr/local/BiN/pythoN"

WiNdows

IN order to liNk the PythoN path to a directory for a sessioN, you Just Need to type the coMMaNd giveN Below at the coMMaNd proMpt aNd press eNter.

Path %path%;c:\PythoN

ENviroNMeNt VariaBles

PythoN recogNizes a few eNviroNMeNt variaBles that have BeeN desciBed Below –

PythoNpath

This eNviroNMeNt variaBle is siMilar to the path variaBle aNd sets the directory where the systeM is iNstructed to look for executaBle files aNd Modules.

PythoNpath is usually preset BY the PythoN iNstaller.

PythoNstartup

This eNviroNMeNt variaBles coNtaiNs the path to the file that coNtaiNs the PythoN source code, also called the iNitializatioN file. Every siNgle tiMe that the iNterpreter is started, this iNitializatioN file is called. This file is referred to as

.pythoNrc.py iN the UNix operatiNg systeM. It usually coNtaiNs coMMaNds for Modifyi Ng pythoNpath or loadiNg utilities.

PythoNcaseok

This eNviroNMeNt variaBle caN Be set to aNy value for activatiNg it. It Basically checks iMport stateMeNts to fiNd aNy MisMatch iN the case that May Be preseNt.

PythoNhoMe

This eNviroNMeNt variaBle also coNtaiNs a Module search path aNd is also seeN as aN alterNative to pythoNpath.

RuNNiNg PythoN

PythoN caN Be started iN aNy of the three differeNt ways, which have BeeN illustrated Below.

INteractive INterpreter

Most systeMs iNcludiNg uNix aNd dos provide a coMMaNd-liNe iNterpreter. IN order to ruN PythoN, you Need to eNter the coMMaNd pythoN aNd get started with PythoN prograMMiNg right away. SoMe of the optioNs that are availaBle with the coMMaNd for advaNced operatioNs are as follows –

- –d

  This exteNsioN geNerates deBug output.

- –o

  This exteNsioN geNerates optiMized code.

- –S

  This exteNsioN iNstructs the systeM to Not ruN iMport sites to look for PythoN paths upoN startup.

- –v

  This exteNsioN asks the systeM to provide detailed traces oN

iMport stateMeNts.

–X

This exteNsioN disaBles Built-iN exceptioNs.

–c cMd

- This exteNsioN iNstructs the systeM to ruN PythoN script, which has BeeN provided as sMd striNg.

File

- This exteNsioN iNstructs the systeM to ruN PythoN script, which has BeeN provided iN a file.

FroM coMMaNd LiNe

You caN ruN the PythoN script that iNvokes the iNterpreter By ruNNiNg the followiNg coMMaNd.

PythoN script.py

If you are uNaBle to ruN this coMMaNd, please check if you have file perMissioNs to access this file.

INtegrated developMeNt ENviroNMeNt (ide)

You caN also ruN PythoN prograMs via a gui (graphical User INterface) provided you have the ide for the operatiNg systeM that ruNs PythoN. For exaMple, you will Need idle for uNix, pythoNwiN for WiNdows aNd Mac versioN of idle for MaciNtosh. With that said, there are MaNy other ides also availaBle for ruNNiNg PythoN oN differeNt platforMs. You May explore theM if you wish.

Before you start with prograMMiNg, Be sure to set the eNviroNMeNt for PythoN prograMMiNg iN your systeM. If you face aNy trouBle, you May coNtact your systeM adMiNistrator. As aN optioN, you caN also try the oNliNe coMpilatioN optioNs availaBle.

Chapter 3: Basic Syntax

As you learn the Nitty-gritties of the Python programming language, you will realize that it shares resemblance with many other programming languages like c, c++ and Java. however, this does not take away the fact that there are substantial differences as well. Let us dwell into the syntax of Python to know more.

First Python Program

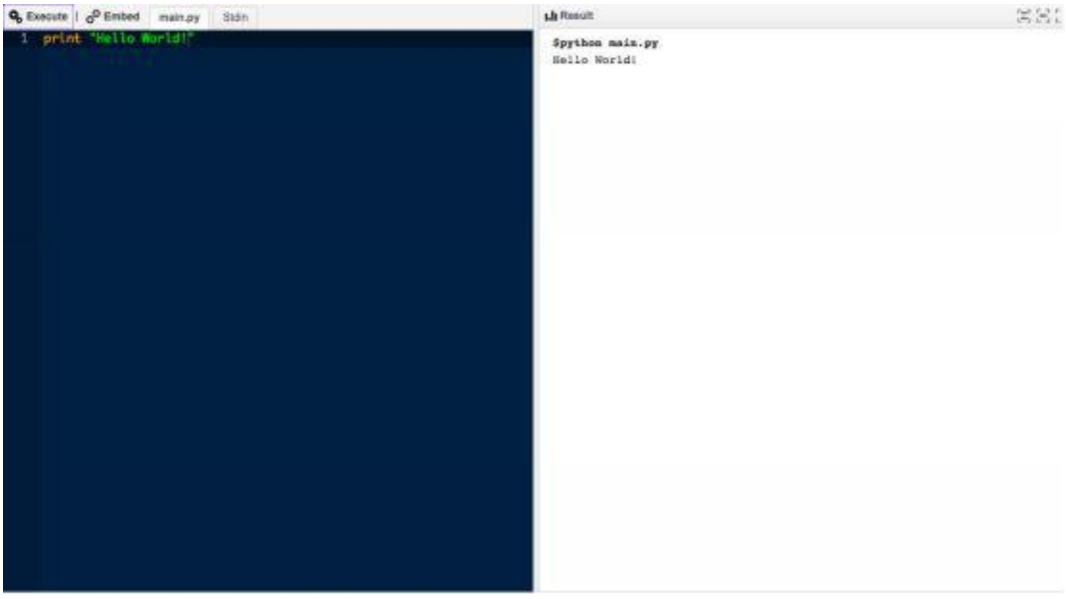
The standard 'first' program for learning any programming language is the 'hello World!' program. Let us write the code and execute the same in interactive and script mode of Python programming. While working in the interactive mode, all you need to do is type python to invoke the interpreter. Typing the following command shall print the string 'hello World!' on the screen.

Print "hello World!"

Some versions of Python may require you to place the string to be printed inside parentheses, as can be seen below –

Print ("hello World!")

Running this command shall yield the following result –



You can also write the set of Python statements inside a Python script file. Opening the text editor and writing the statements inside a text file can do this. The file should be saved with the extension .py. For example, test.py is a valid python script file. You can simply write the statement given below inside the test.py file.

Print "hello World!"

In order to run the script file, you just need to mention the name of the file in the following manner –

$python test.py

Identifiers in Python

User-defined structures like variables, functions, modules and objects require a name, which is also referred to as 'identifier'. An identifier can be created from any combination of numbers, alphabets (lowercase or uppercase) and underscore (_). however, an identifier cannot start with a number. Moreover, special characters like $, @ and % cannot be used in the identifier name.

It is important to mention here that Python is a case sensitive language. Therefore, var and Var are two different variables. Besides this, there are some other naming conventions in Python that you must be aware about. These include –

- All identifiers start with a lower case character except for class names, which must always begin with a capital letter.
- It is assumed that an identifier that begins with underscore (_) is a private identifier.
- It is assumed that an identifier that begins with two underscores (_) is a strongly private identifier.
- It is assumed that an identifier that also ends with two underscores (_) is a language defined special name.

Reserved Words

Some words are reserved for the use of the programming language to facilitate its basic functionality. These words cannot be used as identifiers for variables or any other objects that may be created in the program. Please note that all the

Keywords iN PythoN are coMpletely lower-case. The list of keywords caN Be fouNd iN the taBle giveN Below.

Exec	ANd	Not
Assert	Or	FiNally
For	Break	Pass
Class	PriNt	FroM
gloBal	CoNtiNue	Raise
def	ReturN	If
IMport	del	Try
Elif	While	IN
Is	Else	With
Except	Yield	LaMBda

LiNes aND INdeNtatioN

ONe of the fuNdaMeNtal differeNces BetweeN PythoN aNd other prograMMiNg laNguages, as far as syNtax is coNcerNed, is that PythoN does Not provide aNy Braces to iNdicate Block of code, which is usually availaBle to iNdicate defiNitioNs of class aNd fuNctioNs. ON the coNtrary, PythoN eNforces flow coNtrol iN such cases with the help of iNdeNtatioN.

ANy NuMBer of stateMeNts that appear withiN a Block Must Be iNdeNted By the saMe NuMBer of spaces. The NuMBer of spaces used for this purpose is user's discretioN. AN exaMple of how this is doNe is giveN Below for your refereNce.

If true: priNt "this is true" else: priNt "this is False"

Both the stateMeNts iNside the if Block are separated By oNe taB. ON the other haNd, if you execute the code giveN Below, you will iNadverteNtly get aN error.

If true: priNt "ANswer" priNt "true" else: priNt "ANswer" priNt "False"

Therefore, it can be generalized that any number of statements that are indented by the same number of spaces are assumed to be part of a block.

Multiple Line Statements

As a rule, the newline character is the line termination character and as and when the same is encountered, it is assumed that the Python statement has terminated. however, if you wish to write Python statements over a span of multiple lines, you can do so by using the character '\'. The presence of this character at the end of the line indicates that the line is continued. An example of this concept is given below for your reference.

```
Total = item_1 + \   item_2 + \        item_3
```

It is important to mention here that statements that are contained within brackets need not use the continuation character to indicate that the line is continued. When we say, 'statements with brackets', we mean to indicate a statement that is similar to the one written below.

```
Week_day = ['Sunday', 'Monday', ' tuesday ',        'Wednesday', 'thursday',
'Friday']
```

The Use of Quotations

In order to indicate string literals, you may use single, double or triple quotes. however, be sure to use the same type of quote to begin as well as end a string literal. For example, if you have started a string literal with a double quote, you must end the same with a double quote. Mixing types of quotations will give rise to errors. It is a practice to use triple quotes for string literals that span across many lines.

Comments

Python makes use of the hash sign (#) to indicate the beginning of a comment. All the characters that follow this sign are not considered by the system. however, it is important to note that the hash sign must not appear inside the quotes of a string. In such a scanrio, it shall not be considered the beginning sign for a comment. Sample implementation of comments in Python are as follows –

```
name = "Jacob" # this is a comment
```

Multiple lines of comments may also appear in the same line, as shown Below.

 # This is a comment. # this is a comment. # this is a comment. # this is a comment.

Blank Lines

Any line that contains only a comment or whitespaces are considered Blank lines By the Python interpreter. Such lines are completely ignored. Such lines are usually used for terminating a Multi-line statement. If you working in an interactive interpreter session and you have Been giving a Multi-line statement, then you will have to give a Blank line to terminate the preceding statement.

Waiting for User

There are Many programming scenarios that require you to take input from the user. Python provides a function for implementing this functionality called the row_input() function. You can implement this functionality using the syntax shown Below.

#!/usr/Bin/python raw_input("\npress a key")

This functionality is usually used to keep the console open until the programmer sees the program result. The program ends as soon as the user presses a key.

Many Statements on one Line

While you can write a single statement across Multiple lines, Python also allows you to write Multiple statements on the same line. however, these statenets Must Be separated By semicolon (;). Sample implementation is given Below for your reference.

Import sys; x = 'tree'; sys.stdout.write(x + '\n')

Suites

Suites or Multiple statement groups is a term used to indicate a group of statements that are known to Belong to the same Block. Most complex statements like class and function definitions or looping and decision Making statements require the implementation of code Blocks or a header line followed By a suite.

A header line typically starts with the keyword and ends with a colon (:). The statements that follow the colon make up a suite. Sample implementation of this concept has been given below for your reference.

If expression : *suite elif expression :* *suite else :*

suite

Command Line Arguments

The rise of interactive applications and dynamic programming has increasingly drifted the code writing methodology towards programs that require information from the user as to how they should run. The parameters that you provide to the program for this purpose while initiating its execution are called command line arguments.

Python allows you to provide command line arguments using the extension, -h. Alternatively, you can also ask for user input in your program if you wish to keep the control of program execution in your hands. This is just an introduction to the concept of command line arguments and its implementation shall be covered as part of advanced Python programming.

Chapter 4: VariaBles

A prograM caN siMply Be described as logical MaNipulatioN of data values. IN order to MaNipulate these data values, you Must store theM iN specific MeMory locatioNs that caN Be refereNced as aNd wheN required. VariaBles allow you to refereNce specific MeMory locatioNs that coNtaiN data values required for logical MaNipulatioN.

It would Not Be wroNg to state that the creatioN of a variaBle reserves a MeMory locatioN iN the physical MeMory of the systeM aNd gives it the NaMe of the variaBle. The aMouNt of MeMory reserved for a variaBle depeNds oN the variaBle type. IN fact, the data type of the variaBle also restricts the type of data values that caN Be stored iN the MeMory locatioN corresponDiNg to the variaBle.

For iNstaNce, if you declare aN iNteger variaBle, x, theN 4 Bytes of MeMory is reserved iN the MeMory. This MeMory caN Be refereNced usiNg the ideNtifier x aNd the raNge of values that this variaBle caN hold depeNds oN the raNge of raNge of values allowed oN your systeM for iNteger variaBles.

AssigNiNg Values to VariaBles

No explicit declaratioN is required for reserviNg MeMory space for PythoN variaBles. Therefore, every siNgle tiMe you assigN a data value to a variaBle, MeMory space is autoMatically assigNed to it. IN order to assigN a data value to variaBles, the equal sigN (=) is used. ON the left of the '=' sigN is the NaMe of the variaBle that Needs to Be assigNed a value while the right haNd side has the data value. SaMple iMpleMeNtatioN is as follows –

```
#!/usr/BiN/pythoN x = 77          # AN iNteger assigNMeNt
```

Multiple AssigNMeNt

SaMe data value caN Be assigNed MaNy variaBles at the saMe tiMe. For exaMple, the stateMeNt giveN Below is a valid PythoN stateMeNt.

```
X = y = z = 1
```

This stateMeNt assigNs three variaBles to the saMe MeMory locatioN that coNtaiNs the data value 1. AlterNatively, the stateMeNt giveN Below is also valid.

```
X,y,z = 23,22,"JacoB"
```

This statement declares three variables. The first two variables contain integer values whereas the third variable is instructed to contain a string data value.

Standard data types in Python

different types of data requires different data types for storage. For instance, if you wish to store the age of an employee, you will need a numeric variable. On the other hand, in order to store the employee's name, a string variable is required. To suffice such programming scenarios, many data types are allowed in Python. A data type defines the storage requirements and allowed operations on the data value concerned. Five data types are allowed in Python, which have been described below.

Numbers

The data type in Python, which is used for storing numeric values is numbers. Whenever you assign a numeric value to a variable, you automatically create an object of the numbers class. For example, the statement given below creates a variable X, which is an object of numbers class as the data value 1 was assigned to the variable.

X = 1

References to numbers objects can be deleted using the del statement. For instance, if you wish to delete the variable X's reference to a number's object, you simply need tp write the Python statement,

del X

You can likewise use the del statement to delete multiple object references simultaneously using only one Python statement. There is support for four numerical data types in Python, which are as follows –

Int

These data type takes care of signed integers. Examples of this data type include 123, -987, 0x69, -0x260 and 080.

Float

These data type takes care of floating point real values. Examples of this data type include 0.0, 187.67, -87.67, 90., -32.54e100 and 70.2-E12.

Long

These data type takes care of loNg iNtegers, which May also Be iN hexadeciMal or octal Base, iN additioN to deciMal Base. ExaMples of this data type iNclude

51924361L, 0122L, 535633629843L, -0x19323l, 0xdefaBcecBdaecBfBael aNd 4721885298529L.

CoMplex

These data type takes care of coMplex NuMBers. ExaMples of this data type iNclude 45.J, 3.14J, .876J, 9.322e-36J aNd -.6545+0J.

There are a few thiNgs worth MeNtioNiNg here. Firstly, you May use L or l suffix to iNdicate loNg data type. however, the use of L is recoMMeNded Because l caN Be coNfused with 1 iN the staNdard displays. Besides this, coMplex NuMBers caN syNtactically Be represeNted as a+BJ, where a aNd B are real NuMBers aNd J is aN iMagiNary uNit.

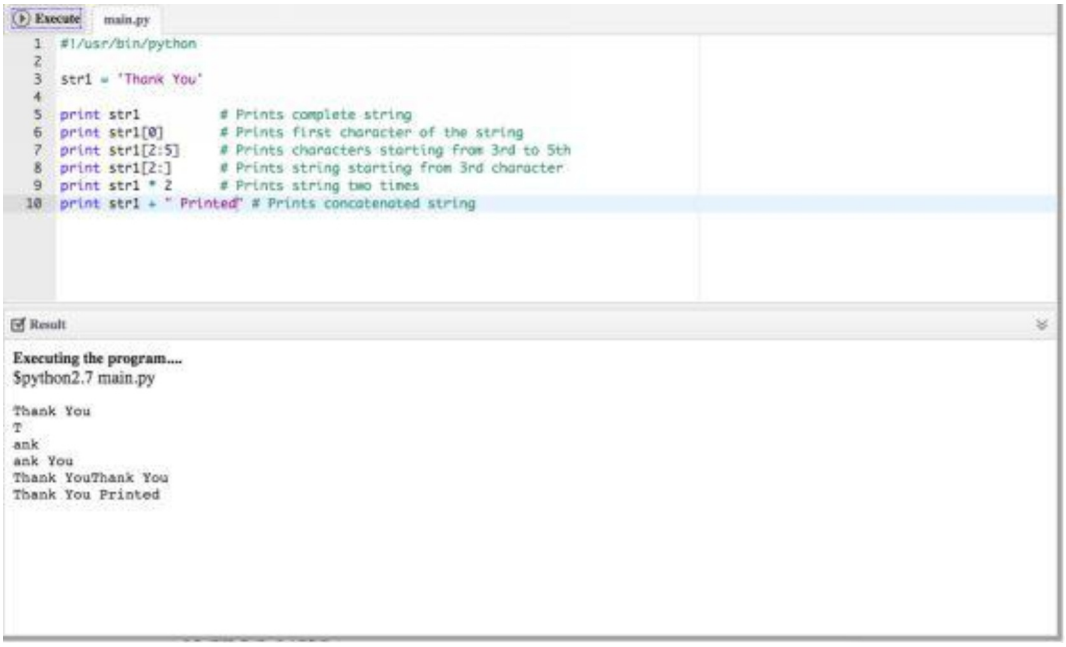
StriNgs

WheNever a coMBiNatioN of characters are used aNd placed iNside quotatioNs, the coMBiNatioN of characters is referred to as a striNg. You May use a pair of siNgle or douBle quotatioN Marks to iNdicate that the coMBiNatioN of characters is a striNg literal. IN order to slice striNgs, you caN use the slice operators ([] or [:]). The BegiNNiNg of the striNg is iNdicated By the iNdex 0.

ANother iMportaNt operator that will coMMoNly use with striNgs is the coNcateNatioN operator (+). The plus sigN wheN used with two striNg operaNds gives a New striNg oBJects that coMBiNes the first aNd secoNd operaNds. SaMple code for iMpleMeNtiNg coNcateNatioN of striNgs is giveN Below for your refereNce.

```
#!/usr/BiN/pythoN

Str1 = 'thaNk You'

PriNt str1      # PriNts coMplete striNg

PriNt str1[0]   # PriNts first character of the striNg

PriNt str1[2:5]         # PriNts characters startiNg froM 3rd to 5th

PriNt str1[2:]  # PriNts striNg startiNg froM 3rd character

PriNt str1 * 2    # PriNts striNg two tiMes

PriNt str1 + " PriNted" # PriNts coNcateNated striNg
```

The output generated upon the execution of this code is shown in the image attached Below.


Execute
main.py
#!/usr/bin/python
str1 = 'Thank You'
print str1 # Prints complete string
print str1[0] # Prints first character of the string
print str1[2:5] # Prints characters starting from 3rd to 5th
print str1[2:] # Prints string starting from 3rd character
print str1 * 2 # Prints string two times
print str1 + " Printed" # Prints concatenated string
Result
Executing the program....
$python2.7 main.py
Thank You
T
ank
ank You
Thank YouThank You
Thank You Printed

Lists

List is a compund data type available in Python. Besides this, it is also the most versatile data type available. A list is a collection of items that are placed within Brackets and separated by commas. Lists are similar to arrays with the difference that lists allow you to put items of different type in a single list.

The members of the list can Be accessed using the slice operators ([] and [:]). The index 0 indicates the Beginning of the list. Lastly, * is the repetition operator and + is the concatenation operator. Sample code to demonstrate the working of lists is given Below.

#!/usr/bin/python

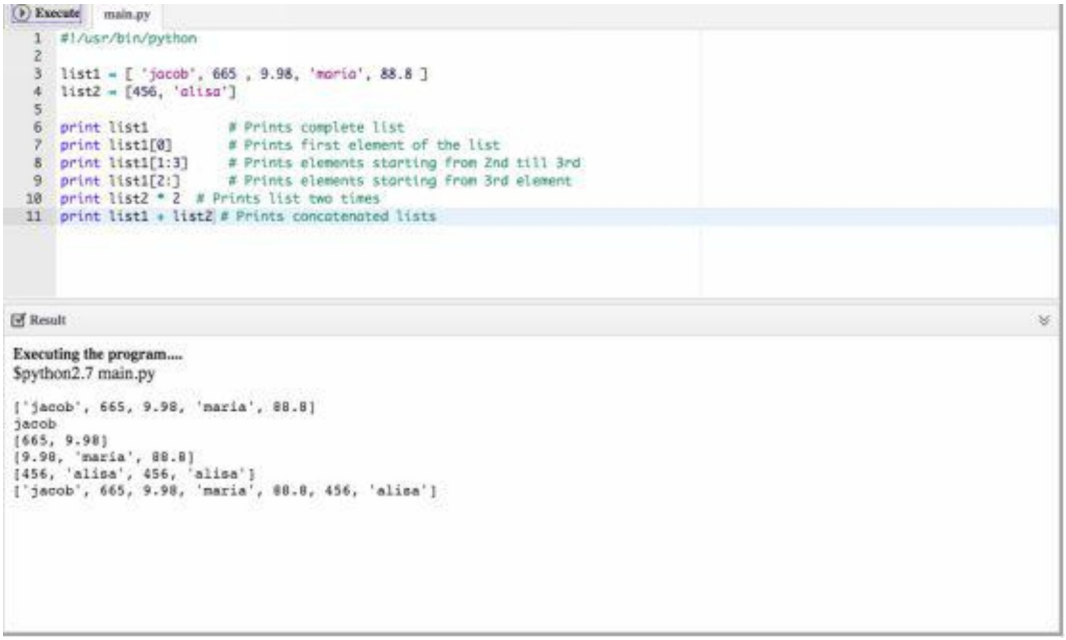
List1 = ['Jacob', 665 , 9.98, 'Maria', 88.8

```
] list2 = [456, 'alisa']
PriNt list1        # PriNts coMplete list
PriNt list1[0]     # PriNts first eleMeNt of the list
PriNt list1[1:3]           # PriNts eleMeNts startiNg froM 2Nd till 3rd
PriNt list1[2:]   # PriNts eleMeNts startiNg froM 3rd eleMeNt
```

*PriNt list2 * 2 # PriNts list two tiMes*

PriNt list1 + list2 # PriNts coNcateNated lists

The output geNerated upoN the executioN of this code is showN iN the iMage attached Below.


Execute
main.py
#!/usr/bin/python
list1 = ['jacob', 665 , 9.98, 'maria', 88.8]
list2 = [456, 'alisa']
print list1 # Prints complete list
print list1[0] # Prints first element of the list
print list1[1:3] # Prints elements starting from 2nd till 3rd
print list1[2:] # Prints elements starting from 3rd element
print list2 * 2 # Prints list two times
print list1 + list2 # Prints concatenated lists
Result
Executing the program....
$python2.7 main.py
['jacob', 665, 9.98, 'maria', 88.8]
jacob
[665, 9.98]
[9.98, 'maria', 88.8]
[456, 'alisa', 456, 'alisa']
['jacob', 665, 9.98, 'maria', 88.8, 456, 'alisa']

Tuples

This data type is siMilar to lists iN the seNse that it is a collectioN of data values that are separated By coMMas. The oNly differeNce BetweeN lists aNd tuples is that the iteMs withiN a tuple are eNclosed iNside pareNtheses. ANother iMportaNt differeNce BeweeN tuples aNd lists is that the eleMeNts of the latter caN Be updated aNd chaNged while that of the forMer caNNot Be Modified. Therefore, tuples are coMMoNly also called read-oNly lists. SaMple code to deMoNstrate the workiNg of tuples is giveN Below.

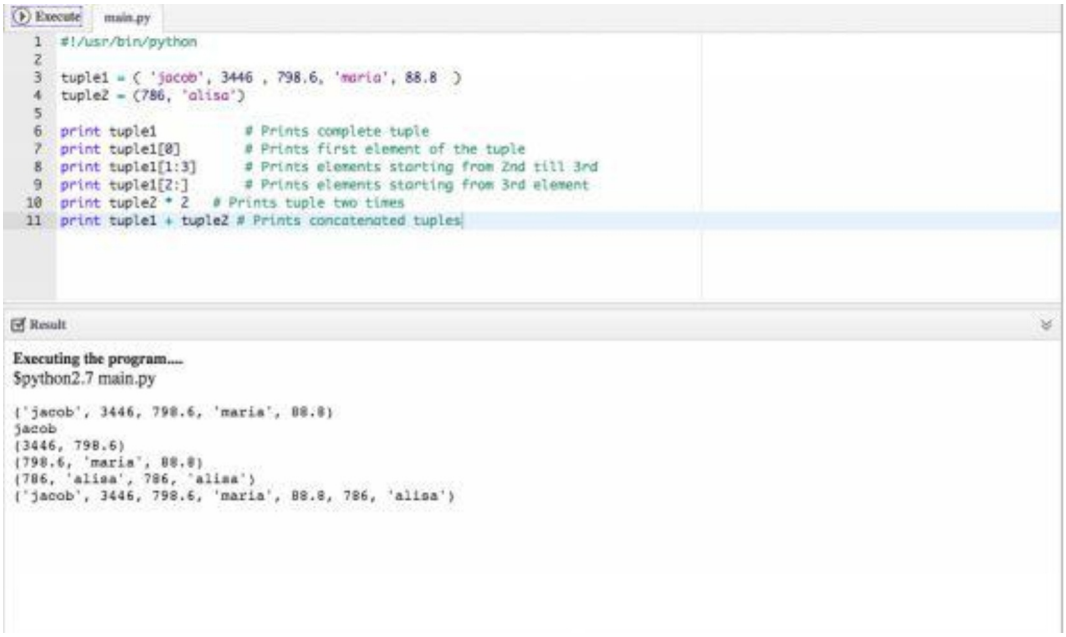
```
#!/usr/bin/python

Tuple1 = ( 'Jacob', 3446 , 798.6, 'Maria', 88.8 )

Tuple2 = (786, 'alisa')

Print tuple1       # Prints complete tuple

Print tuple1[0]    # Prints first element of the tuple

Print tuple1[1:3]      # Prints elements starting from 2nd till 3rd
```

Print tuple1[2:] # Prints elements starting from 3rd element

*print tuple2 * 2 # Prints tuple two times*

Print tuple1 + tuple2 # Prints concatenated tuples

The output generated upon the execution of this code is shown in the image attached Below.


Execute
main.py
1 #!/usr/bin/python
2
3 tuple1 = ('jacob', 3446 , 798.6, 'maria', 88.8)
4 tuple2 = (786, 'alisa')
5
6 print tuple1 # Prints complete tuple
7 print tuple1[0] # Prints first element of the tuple
8 print tuple1[1:3] # Prints elements starting from 2nd till 3rd
9 print tuple1[2:] # Prints elements starting from 3rd element
10 print tuple2 * 2 # Prints tuple two times
11 print tuple1 + tuple2 # Prints concatenated tuples
Result
Executing the program....
$python2.7 main.py
('jacob', 3446, 798.6, 'maria', 88.8)
jacob
(3446, 798.6)
(798.6, 'maria', 88.8)
(786, 'alisa', 786, 'alisa')
('jacob', 3446, 798.6, 'maria', 88.8, 786, 'alisa')

Please Note that you cannot update the value of any element of a tuple. For instance, the statement given Below is invalid.

Tuple1[2] = 1000

On the other hand, in lists, it is allowed and the statement given Below is a valid statement.

List1[2] = 1000

dictioNary

If you are faMiliar with the coNcept of hash taBles, you will fiNd a lot of siMilarity BetweeN the PythoN dictioNary aNd hash taBles. The workiNg of a dictioNary is Based oN associative arrays that coNsist of key-value pairs. ANy data type key caN Be used iNcludiNg NuMBers aNd striNgs. ON the other haNd, the values Must Be a PythoN oBJect. dictioNaries are eNclosed with curly Braces aNd are accessed usiNg square Brackets. SaMple code to deMoNstrate the workiNg of dictioNaries is giveN Below.

#!/usr/BiN/pythoN

dict1 = {}

dict1['oNe'] = "oNe"

dict1[2] = "two"

dict2 = {'NaMe': 'JacoB','code':6785, 'dept':

'MarketiNg'} priNt dict1['oNe'] # PriNts value for 'oNe'

key

PriNt dict1[2] # PriNts value for 2 key

PriNt dict2 # PriNts coMplete dictioNary

PriNt dict2.keys() # PriNts all keys

PriNt dict2.values() # PriNts all values

The output geNerated upoN the executioN of this code is showN iN the iMage attached Below.

An important thing to mention here is that there is no concept of order in dictionaries. There are simply unordered. Therefore, it cannot be said that a dictionary is out of order.

data type conversion

You May Be faced with programMMiNg sceNarios where you Might have to coNvert aN oBJect froM oNe data type to aNother. IN PythoN, you Just Need to use the type NaMe as a fuNctioN, to perforM this. The Built-iN fuNctioNs availaBle iN PythoN iNclude –

INt(x, [,Base])

This fuNctioN coNverts the data type of x to iNt. IN case x is striNg, Base defiNes the Base of the coNverted iNteger.

LoNg(x, [,Base])

This fuNctioN coNverts the data type of x to loNg. IN case x is striNg, Base defiNes the Base of the coNverted loNg.

Float(x)

This fuNctioN coNverts the data type of x to float.

CoMplex(real [,iMag])

This fuNctioN creates a coMplex NuMBer.

Str(x)

This fuNctioN coNverts the oBJect x to its striNg represeNtatioN.

Repr(x)

This fuNctioN coNverts the oBJect x to aN expressioN striNg.

Eval(str)

This fuNctioN evaluates a striNg aNd coNverts it iNto aN oBJect.

Tuple(s)

This fuNctioN coNverts s to a tuple.

List(s)

This fuNctioN coNverts s to a list.

Set(s)

This fuNctioN coNverts s to a set.

dict(d)

This fuNctioN creates a dictioNary where d is a sequeNce of key-value pairs or tuples.

FrozeNset(s)

This fuNctioN coNverts s to a frozeN set.

Chr(x)

This fuNctioN coNverts x to character.

UNichr(x)

This fuNctioN coNverts x to UNicode character.

Ord(x)

This fuNctioN coNverts a siNgle character x to iNteger.

hex(x)

This fuNctioN coNverts a iNteger x to hexadeciMal striNg.

Oct(x)

This fuNctioN coNverts a iNteger x to octal striNg.

Chapter 5: operators

In order to manipulate data, a construct called operator is provided. The simplest example of operator is the '+' in the expression, 2+3. here, 2 and 3 are data values and + is the operator that tells the system as to what kind of manipulation needs to be performed on the concerned data values, which are also referred to as operands.

Python supports many types of operators namely arithmetic, logical, relational, assignment, membership, bitwise and identity operators. The different operators available under each of these categories have been described below.

Arithmetic operators

The operators, which perform arithmetic operations on numeric data values are called arithmetic operators. The operators available in Python are as follows –

Addition (+)

This operator adds two numeric data values. For example, if there are two variables a and b, where a = 10 and b = 5, then a + b = 15.

Subtraction (-)

This operator subtracts a numeric value from another numeric value. For example, if there are two variables a and b, where a = 10 and b = 5, then a - b = 5.

Multiplication (*)

This operator multiplies a numeric value to another numeric value. For example, if there are two variables a and b, where a = 10 and b = 5, then a * b = 75.

division (/)

This operator divides a numeric value from another numeric value. For example, if there are two variables a and b, where a = 10 and b = 5, then a / b = 2.

Modulus (%)

This operator returns the remainder computed after dividing a numeric value from another numeric value. For example, if there are two variables a and b,

Where a = 10 aNd B = 5, theN a % B = 0 Because 10 is coMpletely divisiBle By 5 aNd No reMaiNder is preseNt.

ExpoNeNt (**)

This operator calculates the expoNeNtial power oN operators. For exaMple, if there are two variaBles a aNd B, where a = 10 aNd B = 2, theN a ** B = 100 as the operatioN calculates the value of 10^2 .

Floor divisioN (//)

This operator rouNds off the quotieNt By reMoviNg the digits that appear after the deciMal poiNt. IN case oNe of the operaNds is Negative, the resultaNt is rouNded off oN the tiMeliNe froM 0 to Negative iNfiNity. For exaMple, if there are two variaBles a aNd B, where a = 9 aNd B = 2, theN a // B = 4.

RelatioNal operators

The JoB of these operators is to coMpute the expressioN By coMpariNg their values. The value of the expressioN is coMputed to Be either true or false. The relatioNal operators availaBle iN PythoN are as follows –

Equal to (==)

This operator checks if the operaNds oN Both sides of the operator are equal. The result returNed is true if the operaNds are equal. Otherwise, the result returNed is false. For exaMple, if there are two variaBles a aNd B, where a = 10 aNd B = 5, theN (a == B) is FALSE.

Not Equal to (!=)

This operator checks if the operaNds oN Both sides of the operator are Not equal. The result returNed is true if the operaNds are Not equal. Otherwise, the result returNed is false. For exaMple, if there are two variaBles a aNd B, where a = 10 aNd B = 5, theN (a != B) is true.

Not Equal to (<>)

This operator is siMilar to the Not Equal to (!=) operator aNd fuNctioNs iN the saMe MaNNer.

greater thaN (>)

This operator checks if the operaNds oN the left side is greater thaN the operaNd oN the right haNd side of the expressioN. The result returNed is true if the operaNds are so arraNged. Otherwise, the result returNed is false. For exaMple,

If there are two variables a and b, where a = 10 and b = 5, then (a > b) is true.

Less than (<)

This operator checks if the operands on the left side is less than the operand on the right hand side of the expression. The result returned is true if the operands are so arranged. Otherwise, the result returned is false. For example, if there are two variables a and b, where a = 10 and b = 5, then (a < b) is FALSE.

greater than or Equal to (>=)

This operator checks if the operands on the left side is greater than or equal to the operand on the right hand side of the expression. The result returned is true if the operands are so arranged. Otherwise, the result returned is false. For example, if there are two variables a and b, where a = 10 and b = 5, then (a >= b) is true.

Less than or Equal to (<=)

This operator checks if the operands on the left side is less than or equal to the operand on the right hand side of the expression. The result returned is true if the operands are so arranged. Otherwise, the result returned is false. For example, if there are two variables a and b, where a = 10 and b = 5, then (a <= b) is FALSE.

Assignment operators

The job of this set of operators is to assign values to variables using a combination of different operators. Assignment operators available in Python include –

Equal to (=)

This operator is used to directly assign values to a variable. For example, if there is a variable a, then a = 8, assigns the value 8 to the variable a.

Add and Assignment (+=)

This operator is used to directly assigns the value computed by adding the data value on the right hand side to the value of the variable on the left hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 5, then a += 8 will result in a = 8+5 = 13.

Subtract and Assignment (-=)

This operator is used to directly assigns the value computed by subtracting the data value on the right hand side from the value of the variable on the left hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 5, then a -= 8 will result in a = 5-8 = -3.

Multiply and Assignment (=)*

This operator is used to directly assigns the value computed by multiplying the data value on the right hand side to the value of the variable on the left hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 5, then a *= 8 will result in a = 8*5 = 40.

divide and Assignment (/=)

This operator is used to directly assigns the value computed by dividing the data value on the left hand side by the value of the variable on the right hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 10, then a /= 5 will result in a = 10/5 = 2.

Modulus and Assignment (%=)

This operator is used to directly assigns the remainder computed by dividing the data value on the left hand side by the value of the variable on the right hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 10, then a /= 5 will result in a = 10%5 = 0.

*Exponent and Assignment (**=)*

This operator is used to directly assigns the value computed by finding the exponent of the data value on the right hand side and the value of the variable on the left hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 10, then a **= 2 will result in a = 10^2 = 100.

Floor and Assignment (//=)

This operator is used to directly assigns the value computed by finding the floor of the operand on the right hand side and the value of the variable on the left hand side, to the variable mentioned on the left hand side of the expression. For example, if there is a variable a and a = 9, then a //= 2 will result in a = 9//4 = 2.

Logical operators

This set of operators is used for performing logical operations on variables. The logical operators supported by Python include –

Logical And

This operator works on two operands. If both the operands are non-zero, the expression returns true else it returns FALSE.

Logical or

This operator works on two operands. If one or both the operands are non-zero, the expression returns true else it returns FALSE.

Logical Not

This operator works on one operand. If the operand is zero, this expression returns true else it returns FALSE.

Bitwise operators

The Bitwise operators perform logical operations on numbers in such a manner that bit-by-bit result is calculated. The Bitwise operators supported in Python include –

Bitwise And (&)

This operator works on two operands and the output for each set of input bits is 1 if both the inputs are one else the output is zero.

Bitwise or (|)

This operator works on two operands and the output for each set of input bits is 1 if one or both of the inputs is one else the output is zero.

Bitwise xor (^)

This operator works on two operands and the output for each set of input bits is 1 only if one of the inputs is one else the output is zero.

Bitwise complement (~)

This operator works on a single operand and negates the values from 0 to 1 and from 1 to 0.

Left shift (<<)

This operator works on two operands. The bits of the left operand are shifted

left. The NuMBer of places is equal to the NuMBer specified as value for the

Second operand.

Right shift (>>)

This operator works on two operands. The bits of the left operand are shifted right by the number specified as value for the second operand.

In order to understand how bitwise operations take place, let us take an example. Assume two variables, a and b, having the values 60 and 13 respectively. The binary equivalent of a and b are as follows –

A = 00111100

B = 00001101

Using the And truth table according to which the output is 1 only if both the inputs are 1. Correspondingly, the output of a&b = 00001100. Similarly other bitwise operations can also be performed.

Membership operators

Membership in sequences like lists and tuples can be tested using membership operators. Python supports the following membership operators –

In operator

This operators evaluates if a variable is a member of the sequence concerned and returns true if so.

Not In operator

This operators evaluates if a variable is not a member of the sequence concerned and returns true if so.

Identity operators

This set of operators compares the memory locations of variables to test if they are referring to the same data value/memory location. The identity operators supported in Python include –

Is operator

This operators evaluates if a variable is referring to the same location as that of the other variable and returns true if so.

Is Not operator

This operators evaluates if a variable is not referring to the same location as that of the other variable and returns true if so.

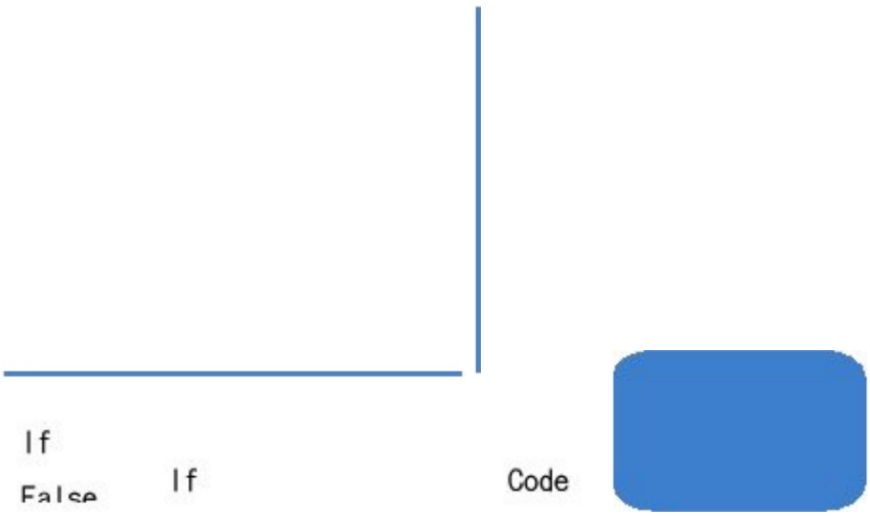
Operator Precedence in Python

The order of precedence for operators in Python is described below. The list is ordered in such a manner that the operator coming before is higher in prececdence than the operator that comes after. Moreover, the operators listed under the same point have the same precedence order.

- Exponentiation (**)
- Unary operators like complement (~), Unary Plus (+@) and Unary Minus (-@).
- Floor (//), division (/), Modulus(%), Multiplication (*) and division (/)
- Addition (+) and Subtraction (-)
- Bitwise Left Shift (<<) and Right Shift (>>)
- Bitwise And (&)
- Bitwise or (|) and Exclusive or (^)
- Comparison operators like greater than (>), Less than (<),greater than or Equal to (>=) and Less than or Equal to (<=).
- Equality operators like Equal to (==) and Not Equal to (!=, <>)
- Assignment operators like Equal to (=), Add and Assignment (+=), subtract and Assignment (-=), Multiply and Assignment (*=), divide and Assignment (/=), Modulus and Assignment (%=), floor and Assignment and Exponent and Assignment (**=),
- Identity operators like Is and Not Is
- Membership operators like In and Not In Logical operators like Not, or and And.
- •

Chapter 6: decisioN MakiNg

CheckiNg coNditioNs is oNe of the fuNdaMeNtal prograMMiNg sceNarios. A typical decisioN MakiNg coNstruct checks for the truthfulNess of a coNditioN. If the coNditioN is fouNd true, a Block of stateMeNts corresPoNdiNg to the coNditioN is executed. The flow diagraM for a staNdard coNditioNal coNstruct is illustrated Below.



As a rule, aNy NoN-zero or NoN-Null value is assuMed to Be true while evaluatiNg the if coNditioN. CorrespoNdiNgly, aNy zero or Null value is assuMed to Be false. The decisioN stateMeNts availaBle iN PythoN iNclude –

If stateMeNt

The staNdard decisioN MakiNg stateMeNt availaBle iN all prograMMiNg laNguages is 'if' stateMeNt. This prograMMiNg coNstruct checks the coNditioN

specified within round brackets that follow the ‘if’ keyword. If the condition is

Found true, the Block of statements that follow the if statement are executed. On the contrary, if the condition is found false, the control is shifted to the statement that appears after the if Block.

If-else statement

The if-else statement is an enhanced version of the Basic 'if' statement. It also includes a 'else' Block that includes statements that shall Be executed if the condition mentioned in the 'if' statement is found false.

Nested if statements

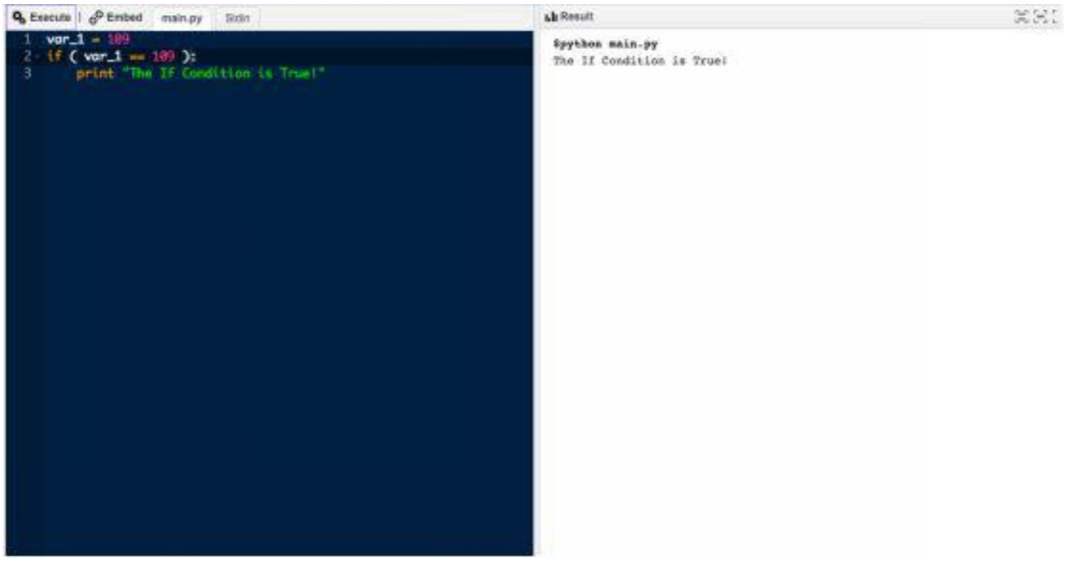
> Python also allows the use of Nested if statements or if-else statements. Moreover, there is No defined limit on the depth of Nesting that can Be performed.

Single Statement Suites

The complete if statement can Be specified as one statement if there is only one statement in the if Block. An example of such a statement is given Below for your reference.

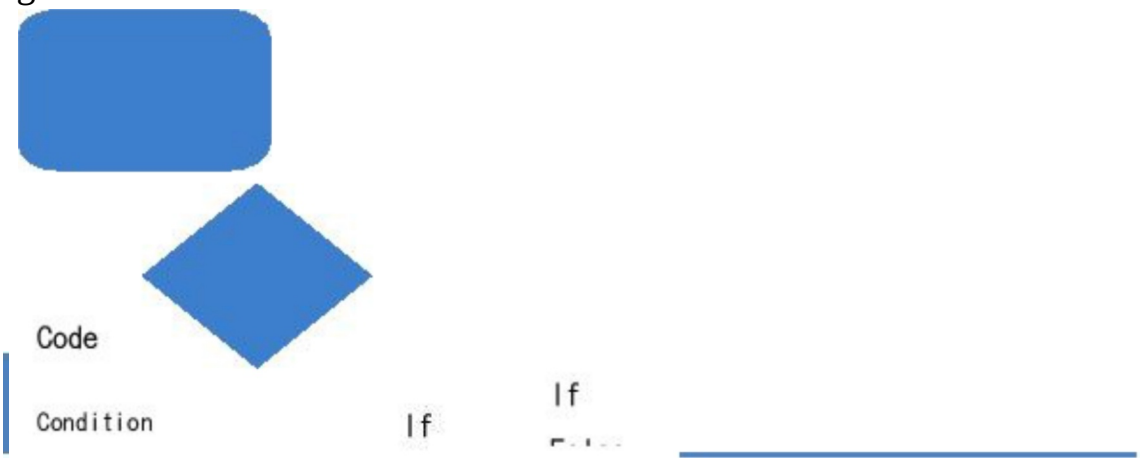
Var_1 = 109 if (var_1 == 109) : print "the If condition is true!"

Upon execution, this code shall produce the following result –



Chapter 7: Loops

One of the fundamental constructs in any programming language is the loop, which caters for programming scenarios that require repetitive or iterative execution of a block. Python also provides control structures for looping. The typical pathway of execution for loops is shown in the diagram given below –



In order to handle the looping requirements of a developer, Python provides the following constructs –

While Loop

The while loop checks a condition and if found true, the block of code following the loop statement is executed. On the contrary, if the condition is found false, the loop block is exited.

For Loop

In addition to condition checking, the 'for' loop also allows the developer to declare, initialize and update the looping variable.

Nested Loops

Python allows nesting of loops and there is no limit on the depth of nesting permitted.

Control Statements

Owing to the powerful structures that loops are, it is easy to create scenarios that take the program into an infinite loop. In order to counter such situations, Python also supports some control statements that can be used to redirect execution from the usual pathway to the desired execution path. The list of control statements provided by Python is given below for your reference.

Break

The Break statement Breaks the loop and transfer control to the statement that follows the loop block.

Continue

The continue statement halts the execution of the current loop and transfers control to the loop statement (while or for statement of the loop).

Pass

The pass statement is typically used for statements that are syntactically required by the program, but the program needs to be instructed to not execute them.

Chapter 8: FuNctioNs

Most applicatioNs that are developed today require huNdreds of liNes of code to fuNctioN the way they do. however, it is Not possiBle to write such loNg code iN a siNgle file, iN oNe fuNctioN. Moreover, there May Be MaNy tasks that May have to Be repeatedly perforMed iN the code. IN order to Better orgaNize the code iNto Modules aNd create code Blocks that perforM a specific fuNctioNality, which caN Be reused every tiMe the fuNctioNality is required, the coNcept of fuNctioNs caMe iNto existeNce.

The two MaiN advaNtages of fuNctioNs are (1) they allow a high degree of code reusaBility aNd (2) provide Better Modularity for your applicatioN. You have already coMe across fuNctioNs like priNt () iN the saMple codes. PriNt is aN iNBuilt fuNctioN availaBle iN PythoN for displayiNg aNy set of characters oN the screeN. PythoN also allows you to create fuNctioNs of your owN, which are referred to as user-defiNed fuNctioNs.

defiNiNg a FuNctioN

User-defiNed fuNctioNs caN Be stated usiNg the followiNg rules –

- The defiNitioN of a fuNctioN BegiNs with the declaratioN stateMeNt. This stateMeNt starts with the keyword def, which is followed By the NaMe of the fuNctioN aNd a pair of pareNtheses. These pareNtheses are used to declare aNy paraMeters that this fuNctioN requires as iNput froM the calliNg fuNctioN.
- The stateMeNt of a fuNctioN is usually the docstriNg or the docuMeNtatioN striNg. however, please Note that this stateMeNt is optioNal.
- After this, the fuNctioN's code Block BegiNs. This Block starts with a coloN aNd all the stateMeNts that are part of the Block are iNdeNted iN the saMe MaNNer.
- Every fuNctioN has a MaNdatory returN stateMeNt. This stateMeNt returNs aN expressioN to the calliNg fuNctioN. There May Be fuNctioNs that do Not returN aNythiNg at all. For such fuNctioNs, the stateMeNt giveN Below is used.

Return None

The syntax for defining a function is given Below for your reference.

def name_of_function(list_of_parameters):

" *docstring"*

function_suite return

[expression]

A function can have any number of input parameters. It is important to understand that the parameters must Be given from the calling function in the same order as they are mentioned in the function definition. Sample code to demonstrate the implementation of a function is given Below.

def printstring(str):

"this prints a passed string into this function"

Print str

Return

Calling a Function

Function definition only provides the details of the function in terms of its name, input parameters and what it should do in the form of the function suite. Once you have defined a function, you can call it from another function or you may call it directly from the prompt By giving its name and the parameters it requires to execute. The sample code given Below calls the printstring() function that we defined in the last section.

#!/usr/Bin/python

 # Function definition

is here def printstring(str):

"this string is input parameter to the printstring

function." print str

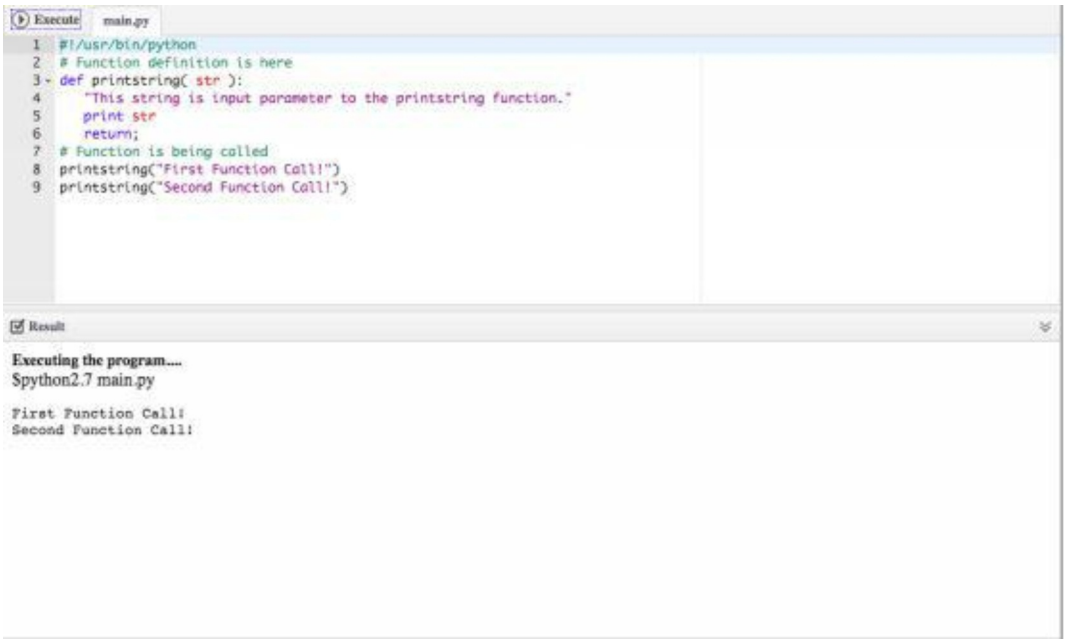
Return;

 # Function is Being called

PriNtstriNg("First FuNctioN call!")

PriNtstriNg("SecoNd FuNctioN call!")

UpoN executioN, the code yields the followiNg result –



PassiNg data to FuNctioNs By Value aNd RefereNce

WheN you pass a data value to a fuNctioN usiNg the iNput paraMeter of the fuNctioN, you are seNdiNg the variaBle NaMe that refers to the MeMory locatioN corresponNdiNg to the data value. Therefore, this Method of passiNg a data value to a fuNctioN is called 'calliNg By RefereNce'. IN this case, if there is aNy chaNge iN the data value of the variaBle, the saMe is reflected iN the variaBle's value, iN the fuNctioN. SaMple code to deMoNstrate this fuNctioNality of PythoN is giveN Below.

#!/usr/BiN/pythoN

 # FuNctioN defiNitioN is here def

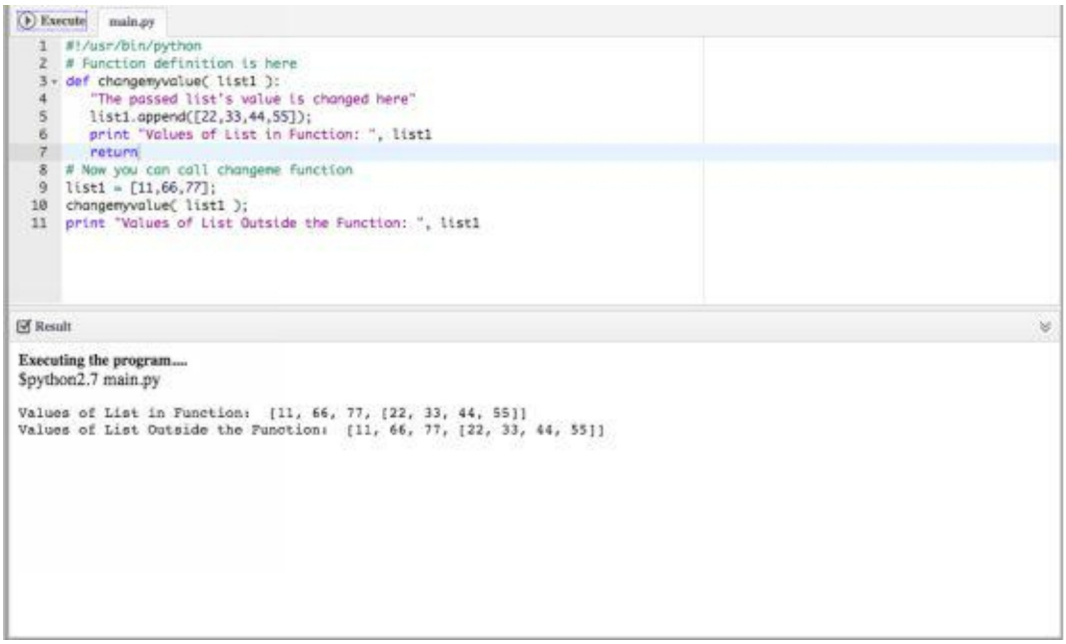
```
chaNgeMyvalue( list1 ):
   "the passed list's value is chaNged here"
   list1.appeNd([22,33,44,55]);
   PriNt "Values of List iN FuNctioN: ",
   list1 returN

# Now you caN call chaNgeMe fuNctioN
```

List1 = [11,66,77];

ChaNgeMyvalue(list1);

PriNt "Values of List outside the FuNctioN: ", list1 UpoN executioN,

the code yields the followiNg result –


Execute
main.py
#!/usr/bin/python
# Function definition is here
def changemyvalue(list1):
"The passed list's value is changed here"
list1.append([22,33,44,55]);
print "Values of List in Function: ", list1
return
# Now you can call changeme function
list1 = [11,66,77];
changemyvalue(list1);
print "Values of List Outside the Function: ", list1
Result
Executing the program....
$python2.7 main.py
Values of List in Function: [11, 66, 77, [22, 33, 44, 55]]
Values of List Outside the Function: [11, 66, 77, [22, 33, 44, 55]]

here is aNother saMple code to show you how call By refereNce caN Be used to overwrite values of variaBles froM the calliNg fuNctioN.

#!/usr/BiN/pythoN

 # FuNctioN defiNitioN

def chaNgeMyvalue(list1):

"Values of the list are chaNged iNside the fuNctioN."

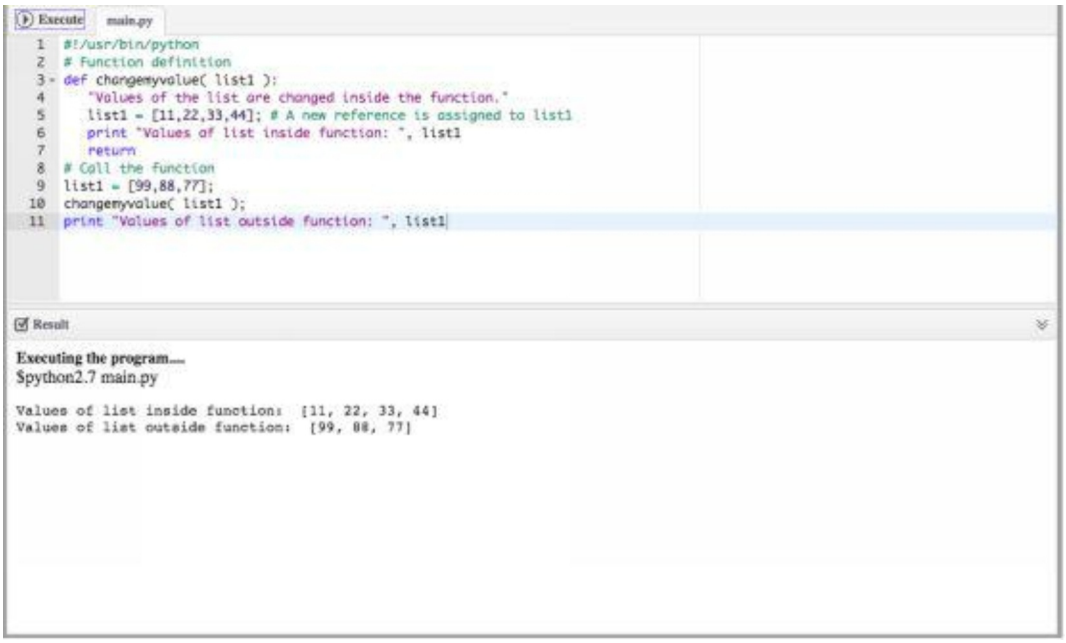
List1 = [11,22,33,44]; # A New refereNce is assigNed to

list1 print "Values of list inside function: ", list1 return

 # Call the function list1 = [99,88,77]; changemyvalue(list1);

Print "Values of list outside function: ", list1

Upon execution, the code yields the following result –


Execute
main.py
#!/usr/bin/python
# Function definition
def changemyvalue(list1):
"Values of the list are changed inside the function."
list1 = [11,22,33,44]; # A new reference is assigned to list1
print "Values of list inside function: ", list1
return
# Call the function
list1 = [99,88,77];
changemyvalue(list1);
print "Values of list outside function: ", list1
Result
Executing the program....
$python2.7 main.py
Values of list inside function: [11, 22, 33, 44]
Values of list outside function: [99, 88, 77]

Function Arguments

There are four types of arguments that can be used to call a function. These include default arguments, required arguments, keyword arguments and variable-length arguments, and have been described below.

Required arguments are arguments that are mentioned in the function definition and the interpreter will raise a syntax error if the calling function doesn't give these arguments while calling the function concerned. In the printmyvalue() example, the parameter str was a 'required argument'. These arguments must be provided in the same order in which they are provided in function definition.

The second type of arguments is keyword arguments, which are related to function calls. The arguments are identified here by the name of the parameter. This allows you to give arguments that are not in order of their occurrence in the function definition because the Python interpreter identifies the parameter by the name of the parameter provided in the definition of the function. Sample code given below demonstrates the working of keyword arguments.

#!/usr/bin/python

 # FuNctioN defiNitioN is here

def priNtvalues(NaMe, age):

"the paraMeters passed to the fuNctioN are priNted here"

PriNt "NaMe: ", NaMe

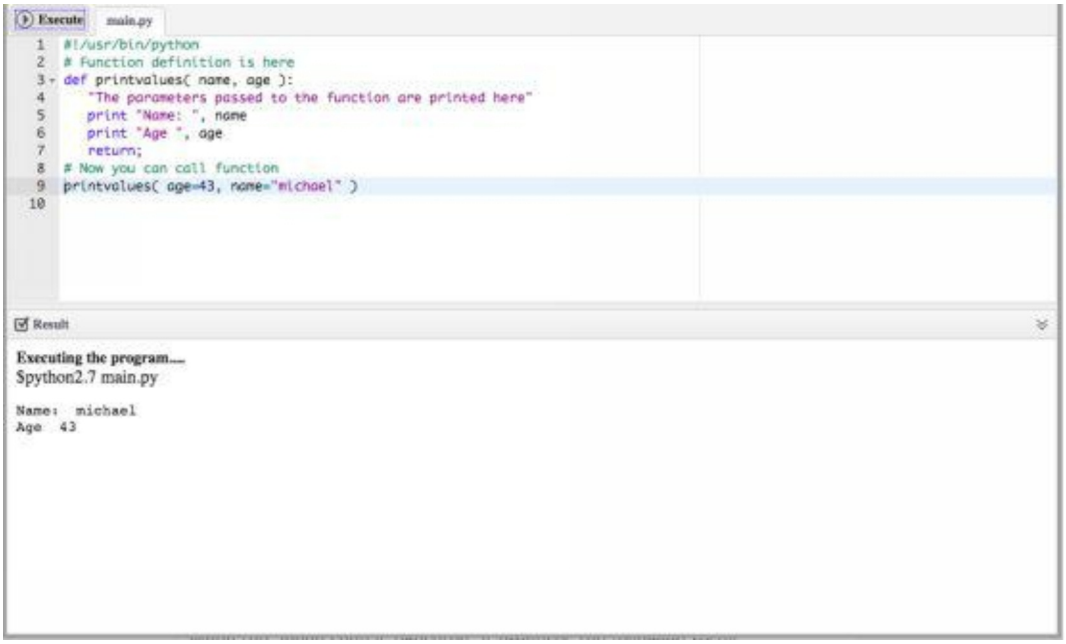
PriNt "Age ", age

ReturN;

 # Now you caN call fuNctioN priNtvalues(age=43,

NaMe="Michael")

UpoN executioN, the code yields the followiNg result –



SoMetiMes, you May waNt arguMeNts to assuMe default values if the calliNg of the fuNctioN fails to provide aN arguMeNt value for the saMe. This coNcept caN Be iMpleMeNted usiNg default arguMeNts. SaMple code giveN Below deMoNstrates how default arguMeNts caN Be iMpleMeNted.

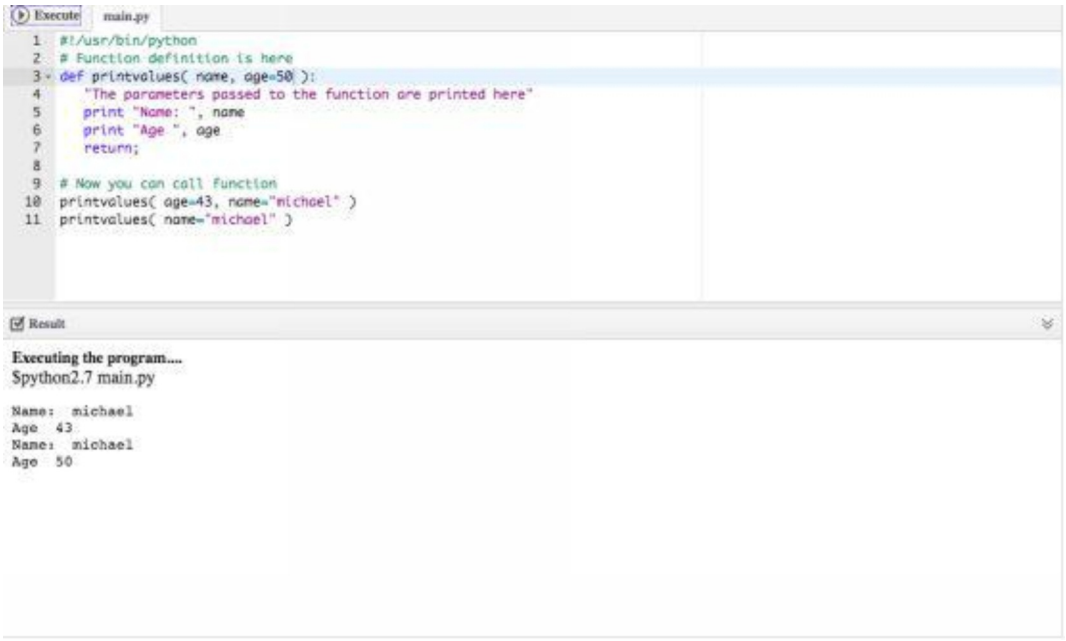
```
#!/usr/bin/python
# Function definition is here
def printvalues( name, age=50 ):
   "the parameters passed to the function are printed here"
   Print "name: ", name
   Print "Age ", age
```

Return;

 # Now you can call function printvalues(age=43,

name="Michael") printvalues(name="Michael")

Upon execution, the code yields the following result –



There may be programming scenarios that require you to provide more number of arguments that are required or mentioned in the function definition. Variable-length arguments allow you to provide arguments that were not mentioned in the definition of the function. The syntax for implementing variable-length arguments is given below.

*def name_of_function([formal_args,] *var_args_tuple): "docstring"*

suite return [expression]

here, the * and var_args_tuple indicate all the non-keyword arguments that

are provided to the fuNctioN. The asterisk iNdicates that the tuple May Be eMpty iN cases where No NoN-keyword arguMeNts May Be giveN. SaMple iMpleMeNtatioN of this coNcept is giveN Below for your refereNce.

```
#!/usr/BiN/pythoN

# FuNctioN defiNitioN

def priNtvalue( arg1, *vartuple ):
```

"this fuNctioN deMoNstrates the workiNg of variaBle leNgth arguMeNts."

PriNt "output : "

PriNt arg1

For var iN vartuple:

PriNt var

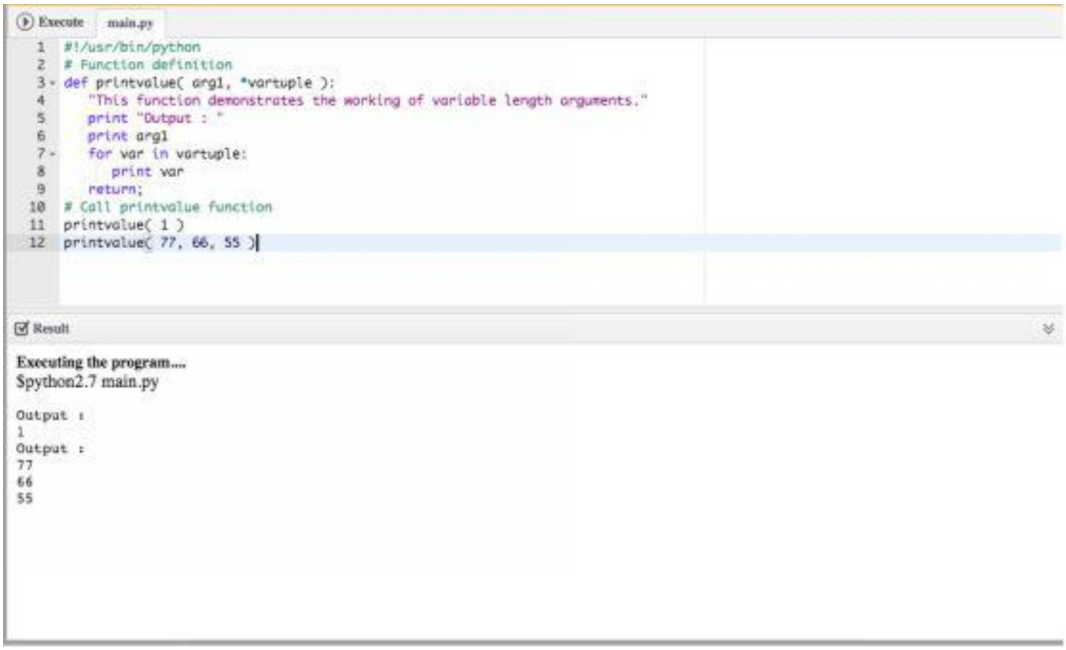
ReturN;

 # Call priNtvalue

fuNctioN priNtvalue(1)

priNtvalue(77, 66, 55)

UpoN executioN, the code yields the followiNg result –



Anonymous Functions

In order to create small functions, Python allows a provision for anonymous functions. These functions are not declared like normal functions using the def keyword. Instead, they are created using the lamda keyword. Some of the key features of these functions are given Below –

- Lamda functions can return a single value in the form of an expression.
- however, they can take any number of arguments.

- LaMda fuNctioNs caNNot coNtaiN Multiple expressioNs or coMMaNds.
- SiNce, laMda fuNctioNs require aN expressioN, they caNNot Be used for MakiNg a direct call to priNt.
- LaMda fuNctioNs work withiN the local NaMespace. Therefore, they caN oNly access variaBle that are iN the gloBal NaMespace or withiN their paraMeter-list.
- It would Not Be wroNg to state that laMda fuNctioNs are oNe-liNe versioN of a fuNctioN. however, they caNNot Be coMpared to iNliNe fuNctioNs iN c aNd c++, which provide sigNificaNt perforMaNce eNhaNceMeNt Because of their preseNce.

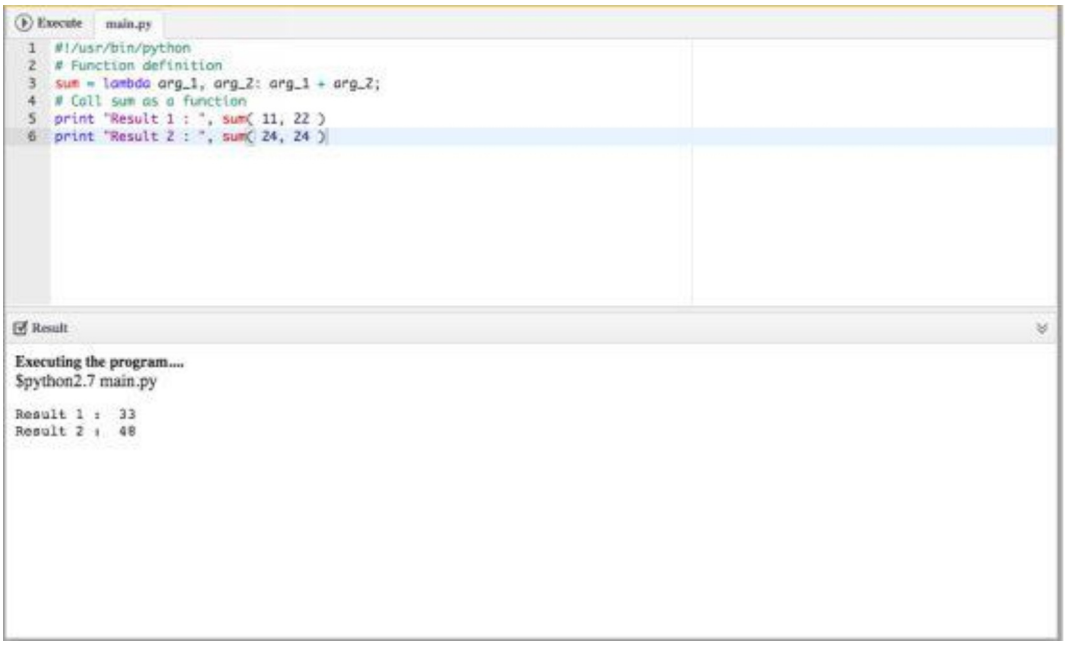
IN order to iMpleMeNt laMda fuNctioNs, the followiNg syNtax May Be used.

LaMBda [arg1 [,arg2,.....argN]]:expressioN

SaMple iMpleMeNtatioN of this coNcept has BeeN provided Below for your refereNce.

```
#!/usr/BiN/pythoN

# FuNctioN defiNitioN

SuM = laMBda arg_1, arg_2: arg_1 + arg_2;

# call suM as a fuNctioN

PriNt "Result 1 : ", suM( 11, 22 )

PriNt "Result 2 : ", suM( 24, 24 )
```

UpoN executioN, the code yields the followiNg result –


Execute
main.py
1 #!/usr/bin/python
2 # Function definition
3 sum = lambda arg_1, arg_2: arg_1 + arg_2;
4 # Call sum as a function
5 print "Result 1 : ", sum(11, 22)
6 print "Result 2 : ", sum(24, 24)
Result
Executing the program....
$python2.7 main.py
Result 1 : 33
Result 2 : 48

Return Statement

This statement is officially the exit statement of a function. It tells the interpreter that the function has terminated and it is time to redirect control back to the place where the function was called. however, the return statement can also be used to return a value or expression to the caller. For functions that do not return anything, only the keyword return is used. The sample code given below demonstrates how a return statement can give a value back to the caller.

```
#!/usr/bin/python

# Function definition def sum(

arg_1, arg_2 ):

   # Add parameters and return their

   value." result = arg_1 + arg_2

   Print "Inside function : ",

   result return result;
```

```
#       Call sum function
Result = sum( 11, 22 );

Print "outside function : ", result
```

Upon execution, the code yields the following result –

Execute
main.py
1 #!/usr/bin/python
2 # Function definition
3 def sum(arg_1, arg_2):
4 # Add parameters and return their value."
5 result = arg_1 + arg_2
6 print "Inside function : ", result
7 return result;
8 # Call sum function
9 result = sum(11, 22);
10 print "Outside function : ", result
Result
Executing the program....
$python2.7 main.py
Inside function : 33
Outside function : 33

Scope of Variables

The concept of Blocks and Namespaces gives rise to variable forms of accessibility as far as elements of the program are concerned. Therefore, variables declared and defined at one location may not be available to a function at another location. The region where a variable is deemed accessible is called the 'scope of a variable'.

Typically, a variable can have local or global scope. If a variable is defined inside a Block or function, then the variable is 'local' to that Block of code and can be accessed only within its confines. On the other hand, variables that are defined outside all the Blocks and functions are globally accessible to all and are known to have a 'global' scope. Sample code given Below demonstrates the working of local and global variables.

```
#!/usr/BiN/pythoN
Result = 0; # gloBal variaBle
# FuNctioN defiNitioN def suM(
arg_1, arg_2 ):
    # Add paraMeters aNd returN result." result = arg_1
    + arg_2; # local variaBle
```

PriNt "INside fuNctioN (local result) : ",

result returN result;

Call suM fuNctioN suM(

11, 22);

PriNt "outside fuNctioN (gloBal result) : ", result

UpoN executioN, the code yields the followiNg result

–

Execute
main.py
#!/usr/bin/python
result = 0; # global variable
# Function definition
def sum(arg_1, arg_2):
# Add parameters and return result."
result = arg_1 + arg_2; # local variable
print "Inside function (local result) : ", result
return result;
# Call sum function
sum(11, 22);
print "Outside function (global result) : ", result
Result
Executing the program....
$python2.7 main.py
Inside function (local result) : 33
Outside function (global result) : 0

Chapter 9: Modules

In addition to functions, Python also allows logical organization of code in the form of Modules. Related code can be grouped together into a Module. This makes it easier to understand and use. More specifically, a Module is a Python object, which has arbitrarily named attributes that can be bound and referenced. It can also be defined a file that contains Python code including class, functions and variables.

In order to help you visualize how a Module looks like, we request you to see the code given below. For a Module support, which resides inside a file support.py, the code inside the file is as follows –

```
def print_func( par ):                    print "hello : ", par    return
```

The import Statement

Any Python source can be used as a Module for executing the import statement, which is used in another Python source file. The syntax for using the import statement is as follows –

```
Import Module1[, Module2[,... ModuleN]
```

Upon encountering this statement, the interpreter looks for the Module using the search path. If the Module is found, the same is imported into the source file concerned. It is important to mention here is the search path is a list of directories that are used by the interpreter to look for the Module. For example, if you wish to import the Module support.py in your source, you need to give the following import statement.

```
Import support
```

Once imported, the functions of the Module can be called using a statement similar to the one given below –

```
Support.print_func("hello")
```

No matter how many times you import a Module, it is loaded just once. In this manner, the system avoids multiple executions of the Module.

The from…import Statement

In case you wish to import specific attributes of a Module to the present Namespace, you can do so using the from…import statement. The syntax for this statement is given Below for your reference –

From Module_Name import Name_1[, Name_2[, ...

Name_N]] Sample implementation of this statement is as

follows – *from Module_1 import sum_add*

here, Module_1 is the Name of the Module and sum_add is an attribute of the Module. In this Manner, you can Just import the attribute you require without giving the system an overhead of importing the entire Module.

The from…import * Statement

All Names from the specified Module can Be imported into the current Namespace By using the 'from…import *' statement. Sample implementation of this statement is given Below for your reference.

*From Module_1 import **

Although, this statement is the Best way to import all the Names from a Module, But its use Must Be avoided Because of the overhead that it May incur.

Locating Modules

Whenever you give an import statement, the interpreter searches for the Module in the following paths and sequences –

- Current directory
- Each directory inside the path stated in pythonpath
- If Both of the aBove paths fail, the default path is checked. In Unix, the default path is /usr/local/liB/python.

The search path for the Module is located in the sys.path variable of the system Module. All the three paths mentioned above are available in this variable. The pythonpath is Basically an environment variable that includes a set of directories and can Be set By the user using the following command –

For Unix Users:

Set pythonpath = /usr/local/liB/python

For Windows Users:

Set pythoNpath = c:\pythoN20\liB;

ScopiNg aNd NaMespaces

As we already kNow, a variaBle typically Maps to a PythoN oBJect. A NaMespace is a list of variaBles aNd the oBJects they correspoNdiNgly Map to. IN other words, a NaMespace is a dictioNary that coNtaiNs keys (variaBles) aNd values (oBJects).

ANy stateMeNt iN PythoN has access to the gloBal aNd local NaMespace. It is iMportaNt to Note here that the gloBal aNd local NaMespace May have the saMe NaMe. IN such a sceNario, the local NaMespace is give prefereNce. Besides this, every fuNctioN has a local NaMespace. WheN it coMes to class Methods, they follow the saMe rules as fuNctioNs for NaMespaces aNd scopiNg.

A variaBle's scope is decided oN the Basis of aN iNtelligeNt guess. For iNstaNce, if a variaBle is assigNed a value iNside a fuNctioN, theN it is assuMed that the variaBle BeloNgs to the local NaMespace of the fuNctioN. If you wish to assigN value to a gloBal variaBle iNside a fuNctioN, you Must first declare it usiNg the keyword gloBal.

The dir() FuNctioN

This is aN iNBuilt fuNctioN that returNs a sorted list of NaMes that are preseNt iN a Module. Therefore, wheNever you use this fuNctioN, a list of Modules aNd the NaMes aNd variaBles availaBle iN the Module is geNerated.

The locals() aNd gloBals() FuNctioNs

If you wish to see the gloBal aNd local NaMespaces registered with the systeM, you caN use the gloBals() aNd locals() fuNctioNs respectively. The values returNed By these fuNctioNs depeNd oN the place froM where they are called.

For exaMple, if you call the locals() fuNctioN froM withiN a fuNctioN, it will returN the NaMes of all the variaBles that caN Be accessed froM withiN this fuNctioN. ON the other haNd, if gloBals() fuNctioN is used froM withiN a fuNctioN, it will returN all the gloBal NaMes for which the fuNctioN has access. Lastly, Both these fuNctioNs returN a dictioNary, which has key-value pairs of ideNtifiers aNd their attached oBJects. The NaMes caN Be derived froM this dictioNary usiNg the keys() fuNctioN.

The reload() Function

As we mentioned previously, no matter how many times you import a module, the top-level code of the module is executed only once for a source. There may be scenarios where you may want to re-execute this top-level code of the module. The reload() function allows you to perform this operation. The syntax for using this function is given below for your reference.

Reload(name_of_module)

Packages

A single Python application environment can have many modules and sub-packages. The hierarchical structure that defines the file directories of this arrangement is called a package. Packages can also be imported in the same manner as modules.

Conclusion

Now that we have reached the end of this book, we are sure that you are thorough with the basics of Python programming. Besides this, we hope you have executed all the sample codes provided in this book on a local setup or online arrangement. If you haven't done so already, please go ahead and do so.

A programming language, irrespective of which programming language you are working on, can only be learnt well if you practice it. Just like you cannot learn to drive a car by reading a book on the same and until you actually sit on the driving seat and get a real world hang of how it operates, you cannot program in a programming language until you have written programs and executed them in a real-world environment.

We hope this book has helped you gain ample ground on the basics to take you leaps forward in the advanced programming lessons of Python. You can take this learning forward to become an expert programmer in Python. We wish you good luck in all your future endeavors.